COMSC-210 Lecture Topic 3
Constructors And Square Brackets

Reference
Childs Ch. 3

Constructors
special functions that are called automatically upon
  object *creation*, to initialize data members
cannot be called via programming
without constructors, data members have
  unpredictable values
  ...unless brace initialized

the *default* constructor
  declarations: *no parentheses*
constructors with parameters
  declarations need parentheses
the *copy* constructor
  special purpose: ref Ch.5
cannot use brace initialization with constructors

e.g.: in the class H file

```
template <class T>
class Road
{
  ...
  public:
  Road(); // default constructor
  Road(const T&, const T&);
  ...
};

template <class T>
Road<T>::Road()
{
  length = 0; // T must allow setting to 0
  width = 0; //...cannot be a string!
};

template <class T>
Road<T>::Road(const T& l, const T& w)
{
  length = l;
  width = w;
};
```

can be inline (defined inside the class curly-braces)

can have multiple constructors in any combination
  but with unique *signatures*

allows programmer to do this:
```
  Road road(1000, 10);
```

...instead of this:
```
  Road road;
  road.setLength(1000);
  road.setWidth(10);
```

A StaticArray Class Template
desired application -- like the C++11 STL `array` template
`StaticArray<int, 10> a;` // create array of 10 `int`s
`a[0] = 100;` // allow assignment
`cout << a[0];` // allow access
`for (i=0; i < a.size(); i++)` // traverse

template specification:
```
template <class T, int CAPACITY>
```

public interface:
```
T& operator[](int); // lookup key is an index
T operator[](int) const;
int size() const; // #of keys "in use", initially zero
int capacity() const;
bool containsKey(int) const;
void deleteKey(int); // set "in use" to false
vector<int> keys() const; // all "in use" keys
void clear();
```

advantages: implements "range safety",
  tracks its own capacity, tracks "in use" elements

Data Members
the values array: `T values[CAPACITY];` // static array
the "in use" array: `bool inUse[CAPACITY];` // static array
`CAPACITY` available in all functions, as part of template

Default Constructor
set all `inUse` to false
do NOT initialize `values`...

`operator[]`, setter version
operator[ ] returns a *reference*
add private member `T dummy;`
why we need a "dummy" data member, `T dummy;`
  key out of range (0 to CAPACITY-1)

```
template <class T, int CAPACITY>
T& StaticArray<T, CAPACITY>::operator[](int key)
{
  // if key out-of-range, return dummy
  // set as "in use" -- activate position when "touched"
  return values[key]; // a mutable reference
}
```

`operator[]`, getter version
if key out of range or
  not "in use", return dummy
do NOT set "in use"
return a COPY instead of a reference

Size, Capacity, and Clear
"capacity" is #of values that a data structure can hold

`operator[ ]`

must be member -- not stand alone
used to "look up" something in an object
its one parameter is a "key" to the lookup
e.g., `a[7]` looks up the value whose key is 7
  for arrays, return the value at index 7

adapted to a Road class
  road[0] could return length, road[1] width
  or road["length"] could return length...

`operator[ ]` Getter

`T operator[](int) const` // returns a copy
called only if "host object" is read-only

`operator[ ]` Setter

`T& operator[](int)`// returns a mutable reference
allows `a[i] = 100;`

`operator[ ]` Variation

`T& operator[](string)`
allows `road["width"] = 10;`

```
T& Road::operator[](string s)
{
  if (s == "length") return length;
  if (s == "width") return width;
  return dummy; // no key match
}
```

"size" means #of values "in use"
  count "in use" values
"clear" setter sets *all* "in use" to false
  affects size, not capacity

Check For "Key" In Use, `containsKey`

return false, if parameter key out of range
otherwise return inUse at key

Delete A "Key", `deleteKey`

return, if key out of range
return, if not inUse at key
otherwise set inUse at key to false

The STL `vector` Template

library: `#include <vector>`
declare an empty vector: `vector<int> v;`
add a value: `v.push_back(x);` // where x is an int
traverse: `for (i=0; i < v.size(); i++) ...v[i]...`

Get All Keys: the `keys` Getter

return keys of "in use" values as a vector:
1. create empty vector of `int`s
2. traverse "in use" array, push_back each key in use
3. return the vector

use to look at array contents:

```
StaticArray<int, 10> a;
...
vector<int> keys = a.keys();
for (int i = 0; i < keys.size(); i++)
  cout << "a[" << keys[i] << " = " << a[keys[i]] << endl;
```

How to confirm comparison operators in a test driver:

```
cout "Testing operator=="
Create a copy (y) of an existing object (x)
if x == y
  cout "x==y, as expected"
else
  cout "test failed!"
assert(x==y)

Use a setter to modify y so it does not equal x
if x == y
  cout "test failed!"
else
  cout "x not equal to y, as expected"
assert(!(x==y))

cout "Testing operator<"
Use a setter to modify y so it is greater than x
if x < y
  cout "x<y, as expected"
else
  cout "test failed!"
assert(x<y)
```

```
Use a setter to modify y so it is less than x
if x < y
  cout "test failed!"
else
  cout "y not less than x, as expected"
assert(!(x<y))
```